

Distributed Graph Automata and Verification of Distributed Algorithms

Fabian Reiter
fabian.reiter@gmail.com

May 2014

Abstract. Combining ideas from distributed algorithms and alternating automata, we introduce a new class of finite graph automata that recognize precisely the languages of finite graphs definable in monadic second-order logic. By restricting transitions to be nondeterministic or deterministic, we also obtain two strictly weaker variants of our automata for which the emptiness problem is decidable. As an application, we suggest how suitable graph automata might be useful in formal verification of distributed algorithms, using Floyd-Hoare logic.

Keywords. Graphs, Finite automata, MSO-logic, Distributed algorithms, Verification

Contents

1	Introduction	2
2	Preliminaries	3
2.1	Graphs and Graph Languages	3
2.2	Logic on Graphs	4
3	Distributed Graph Automata	6
3.1	Informal Description	6
3.2	Formal Definitions	8
3.3	Hierarchy and Closure Properties	12
3.4	Equivalence of ADGAs and MSO-Logic	14
3.5	Emptiness Problem of NDGAs	16
3.6	Summary and Discussion	17
4	Verification of Distributed Algorithms	19
4.1	Distributed Programming Language	19
4.2	Verification Method	22
4.3	Prospects and Limitations	24
4.4	Related Work	25

1 Introduction

The regularity of a language of finite words is a central notion in formal language theory. It is often defined as being recognizable by a finite automaton, but many alternative characterizations exist. By several well-known results, mostly from the late 1950s and early 1960s, it is equivalent whether a language can be

- (a) recognized by a (non)deterministic or alternating finite automaton [RS59, CKS81],
- (b) expressed by a regular expression [Kle56],
- (c) generated by a regular grammar [Cho56],
- (d) obtained as a homomorphic preimage of a subset of some finite monoid [Ner58], or
- (e) defined in (existential) monadic second-order logic [Büc60, Elg61, Tra61].

All of these characterizations can be generalized from words to trees in a natural manner, and, quite remarkably, they all remain equivalent on trees (see, e.g., [TATA08]). Hence, the notion of regularity extends directly to tree languages.

In contrast, the situation becomes far more complicated if we expand our field of interest from words or trees to arbitrary finite graphs (possibly with node labels and multiple edge relations). Although some of the characterizations mentioned above can be generalized to graphs in a meaningful way, they are, in general, no longer equivalent. Perhaps the logical approach (e) is the most straightforward to generalize, since the syntax of monadic second-order logic (MSO-logic) on graphs remains essentially the same as on more restricted structures. While on words and trees the existential fragment of that logic (EMSO-logic) is already sufficient to characterize regularity, it is strictly less expressive than full MSO-logic on graphs, as has been shown by Fagin in [Fag75]. Similarly, the algebraic approach (d) has been extended to graphs by Courcelle in [Cou90], and it turns out that MSO-logic is strictly less powerful than his notion of recognizability, which is defined in terms of homomorphisms into finite algebras. A common pattern that emerges from such results is that the different characterizations of regularity drift apart as the complexity of the considered structures increases. In this sense, regularity cannot be considered a well-defined property of graph languages.

To complicate matters even further, the automata-theoretic characterization (a), which is instrumental in the theory of word and tree languages, does not seem to have a natural counterpart on graphs. A word or tree automaton can scan its entire input by a single deterministic traversal, which is completely determined by the structure of the input (i.e., left-to-right, for words, or bottom-to-top, for trees). On arbitrary graphs, however, there is no sense of a global direction that the automaton could follow, especially since we do not even require connectivity or acyclicity.

Another approach, investigated by Thomas in [Tho91], is to nondeterministically assign a state of the automaton to each node of the graph, and then check that this assignment satisfies certain local “transition” conditions for each node (specified with respect to neighboring nodes within a fixed radius) as well as certain global occurrence conditions at the level of the entire graph. The graph acceptors introduced by Thomas, following this principle, turn out to be equivalent to EMSO-logic on graphs of bounded degree. They are a legitimate generalization of finite automata, in the sense that they are equivalent to them and can easily simulate them if we restrict the input to (graphs representing) words or trees. However, on arbitrary graphs, they are less well-behaved than classical finite automata, which is a direct consequence of their equivalence with EMSO-logic. In particular, they do not satisfy closure under complementation, and their emptiness problem is undecidable.

Contribution. In this paper, we attempt to provide an alternative approach to automata theory on finite graphs. Our model, dubbed *distributed graph automaton*, takes inspiration from distributed algorithms and shares some similarities with Thomas’ graph acceptors. More specifically, we also use a combination of local conditions, which are checked by the nodes using information received from their neighborhood, and global conditions, which are checked at the level of the entire graph. However, both types of conditions are much simpler than in Thomas’ model, which allows us to consider graphs of unbounded degree. Nevertheless, we obtain as a main result that our automata are equivalent to full MSO-logic if we equip them with the power of alternation. If, on the other hand, we only allow nondeterminism, then we get a model that is not closed under complementation, and is even strictly weaker than EMSO-logic, but has a decidable emptiness problem. Interestingly, this model is still powerful enough to characterize precisely the regular languages when restricted to words or trees. Hence, this work also contributes to the general observation, made above, that regularity becomes a moving target when lifted to the setting of graphs. Lastly, by further disallowing nondeterminism, we obtain an even weaker model of computation, which we use to illustrate how automata theory on graphs might have an application in formal verification of distributed algorithms.

Structure. The remainder of this article is organized as follows: Some preliminaries on graphs and logic are reviewed in Section 2. Then we introduce the notion of distributed graph automaton and present our results in Section 3. That section is mostly self-contained and constitutes the main part of this paper. Finally, in Section 4, we sketch an adaptation of Floyd-Hoare logic to synchronous distributed algorithms. Although the idea is presented using (the deterministic variant of) our automaton model, it can be generalized to any type of graph automaton that satisfies certain properties.

2 Preliminaries

We begin by fixing the terminology and notation used in this paper.

2.1 Graphs and Graph Languages

Our objects of interest are finite directed graphs with nodes labeled by an alphabet Σ , and multiple edge relations indexed by an alphabet Γ .

2.1.1 Definition (Σ -Labeled Γ -Graph).

Let Σ and Γ be two finite nonempty sets of node labels and edge labels, respectively. A *Γ -graph* is a structure $G = \langle V_G, \langle \xrightarrow{\gamma}_G \rangle_{\gamma \in \Gamma} \rangle$, where

- V_G is a finite nonempty set of *nodes*, and
- each $\xrightarrow{\gamma}_G \subseteq V_G \times V_G$ is a set of directed *edges* labeled by $\gamma \in \Gamma$.

A (node) *labeling* of G is a function $\lambda: V_G \rightarrow \Sigma$. We call the tuple $\langle G, \lambda \rangle$ a *Σ -labeled Γ -graph* and denote it by G_λ .

If Σ and Γ are understood or irrelevant, we refer to G simply as a graph and to G_λ as a labeled graph, or even just as a graph. We do this especially when the alphabets contain only a single “dummy” symbol, which by default shall be the blank symbol \square . If $\Sigma = \{\square\}$, we also identify G_λ with G .

Given a Γ -graph G , we denote by Σ^G the set of all Σ -labeled versions of G , and by $\Sigma^{\hat{\Gamma}}$ (read “ Σ clouded Γ ”) the set of all Σ -labeled Γ -graphs, i.e.,

$$\Sigma^G := \{G_\lambda \mid \lambda: V_G \rightarrow \Sigma\}, \quad \text{and} \quad \Sigma^{\hat{\Gamma}} := \bigcup_{G \in \mathcal{G}(\Gamma)} \Sigma^G,$$

where $\mathcal{G}(\Gamma)$ is the set of all Γ -graphs. Note that this is very similar to the standard notation of formal language theory on words, where Σ^n designates the set of all Σ -labeled versions of a path of length n (i.e., words over Σ of length n), and $\Sigma^* = \bigcup_{n \in \mathbb{N}} \Sigma^n$.

We are only interested in (labeled) graphs *up to isomorphism*. That is, we consider $G_\lambda, G_{\lambda'} \in \Sigma^{\hat{\Gamma}}$ to be equal if there is a bijection between V_G and $V_{G'}$ that preserves the edge relations and node labels.

A *graph language* is a set of labeled graphs. More precisely, L is a graph language if and only if there are finite nonempty alphabets Σ and Γ , such that $L \subseteq \Sigma^{\hat{\Gamma}}$.

By a (node) *projection* we mean a mapping $h: \Sigma \rightarrow \Sigma'$ between two alphabets Σ and Σ' . With slight abuse of notation, such a mapping is extended to labeled graphs by applying it to each node label, and to graph languages by applying it to each labeled graph. That is, for every $G_\lambda \in \Sigma^{\hat{\Gamma}}$ and $L \subseteq \Sigma^{\hat{\Gamma}}$,

$$h(G_\lambda) := G_{h \circ \lambda}, \quad \text{and} \quad h(L) := \{h(G_\lambda) \mid G_\lambda \in L\},$$

where the operator \circ denotes function composition, such that $(h \circ \lambda)(v) = h(\lambda(v))$.

When reasoning about graphs as structural objects, we will follow the usual terminology of graph theory. In particular, given a Γ -graph G and two nodes $u, v \in V_G$, we say that u is an *incoming neighbor* of v , and v an *outgoing neighbor* of u , if $u \xrightarrow{\gamma}_G v$ for some $\gamma \in \Gamma$. In this case we also say that u and v are *adjacent*, and without further qualification the term *neighbor* refers to both incoming and outgoing neighbors. The *neighborhood* of a node is the set of all of its neighbors. A node without incoming neighbors is called a *source*, whereas a node without outgoing neighbors is called a *sink*.

Finally, let us briefly recall some standard graph properties. Consider a graph $G_\lambda \in \Sigma^{\hat{\Gamma}}$. We say that G_λ is *undirected* if for every $u, v \in V_G$ and $\gamma \in \Gamma$, it holds that $u \xrightarrow{\gamma}_G v$ if and only if $v \xrightarrow{\gamma}_G u$. The graph G_λ is (weakly) *connected* if for every nonempty proper subset U of V_G , there exist two nodes $u \in U$ and $v \in V_G \setminus U$ that are adjacent. The node labeling λ constitutes a valid *coloring* of G if no two adjacent nodes share the same label, i.e., $u \xrightarrow{\gamma}_G v$ implies $\lambda(u) \neq \lambda(v)$, for all $u, v \in V_G$ and $\gamma \in \Gamma$. If $|\Sigma| = k$, such a coloring is called a *k-coloring* of G , and any Γ -graph for which a *k-coloring* exists is said to be *k-colorable*. Note that, by definition, a graph that contains self-loops is not *k-colorable* for any k .

2.2 Logic on Graphs

We fix two disjoint, countably infinite sets of variables: the supply of node variables $\mathcal{V}_{\text{node}} = \{u, v, \dots, u_1, \dots\}$, and the supply of set variables $\mathcal{V}_{\text{set}} = \{U, V, \dots, U_1, \dots\}$. Node variables will always be represented by lower-case letters, and set variables by upper-case ones, often with subscripts.

2.2.1 Definition (Monadic Second-Order Formula).

Let Σ and Γ be two finite nonempty alphabets. The set $\text{MSO}(\Sigma, \Gamma)$ of *monadic second-order formulas* (on graphs) over $\langle \Sigma, \Gamma \rangle$ is built up from the atomic formulas

- $\diamond x$ (“ x has label a ”),
- $x \xrightarrow{\gamma} y$ (“ x has a γ -edge to y ”),

- $x = y$ (“ x is equal to y ”),
- $x \in X$ (“ x is an element of X ”),

for all $x, y \in \mathcal{V}_{\text{node}}$, $X \in \mathcal{V}_{\text{set}}$, $a \in \Sigma$, and $\gamma \in \Gamma$, using the usual propositional connectives and quantifiers, which can be applied to both node and set variables. More precisely, if φ and ψ are $\text{MSO}(\Sigma, \Gamma)$ -formulas, then so are $\neg\varphi$, $\varphi \vee \psi$, $\varphi \wedge \psi$, $\varphi \Rightarrow \psi$, $\varphi \Leftrightarrow \psi$, $\exists x(\varphi)$, $\forall x(\varphi)$, $\exists X(\varphi)$, and $\forall X(\varphi)$, for all $x \in \mathcal{V}_{\text{node}}$ and $X \in \mathcal{V}_{\text{set}}$.

We denote by $\text{free}(\varphi)$ the set of variables in $\mathcal{V}_{\text{node}} \cup \mathcal{V}_{\text{set}}$ that occur freely in φ (i.e., not within the scope of a quantifier), and use the notation $\varphi[x_1, \dots, x_m, X_1, \dots, X_n]$ to indicate that at most the variables given in brackets occur freely in φ , i.e., $\text{free}(\varphi) \subseteq \{x_1, \dots, x_m, X_1, \dots, X_n\}$. If $\text{free}(\varphi) = \emptyset$, we also say that φ is a *sentence*.

The truth of an $\text{MSO}(\Sigma, \Gamma)$ -formula φ is evaluated with respect to a labeled graph $G_\lambda \in \Sigma^{\hat{\Gamma}}$ and a variable assignment $\alpha: \text{free}(\varphi) \rightarrow V_G \cup 2^{V_G}$ that assigns a node $v \in V_G$ to each node variable in $\text{free}(\varphi)$, and a set of nodes $S \subseteq V_G$ to each set variable in $\text{free}(\varphi)$. The meaning of atomic formulas is as hinted informally in Definition 2.2.1. In particular, $\diamond_a x$ is satisfied if and only if $\lambda(\alpha(x)) = a$, and $x \xrightarrow{\gamma} y$ is satisfied if and only if $\alpha(x) \xrightarrow{\gamma}_G \alpha(y)$. For composed formulas, satisfaction is defined inductively by the standard semantics of predicate logic. We write $\langle G_\lambda, \alpha \rangle \models \varphi$ to denote that G_λ and α *satisfy* φ . If φ is a sentence, the variable assignment is superfluous, and we simply write $G_\lambda \models \varphi$ if G_λ satisfies φ .

The graph language $L_{\Sigma, \Gamma}(\varphi)$ *defined* by φ with respect to Σ and Γ is the set of all Σ -labeled Γ -graphs that satisfy φ , i.e.,

$$L_{\Sigma, \Gamma}(\varphi) := \{G_\lambda \in \Sigma^{\hat{\Gamma}} \mid G_\lambda \models \varphi\}.$$

Every graph language that is defined by some MSO-sentence is called *MSO-definable*. We denote by \mathcal{L}_{MSO} the class of all such graph languages.

2.2.2 Example (3-Colorability).

Let $\Sigma = \Gamma = \{\square\}$. The following $\text{MSO}(\Sigma, \Gamma)$ -sentence defines the language of 3-colorable graphs.

$$\begin{aligned} \varphi_3^{\text{color}} := & \exists U_{\spadesuit}, U_{\heartsuit}, U_{\clubsuit} \left(\forall u \left((u \in U_{\spadesuit} \vee u \in U_{\heartsuit} \vee u \in U_{\clubsuit}) \wedge \neg(u \in U_{\spadesuit} \wedge u \in U_{\heartsuit}) \wedge \right. \right. \\ & \quad \neg(u \in U_{\spadesuit} \wedge u \in U_{\clubsuit}) \wedge \neg(u \in U_{\heartsuit} \wedge u \in U_{\clubsuit}) \left. \right) \wedge \\ & \quad \forall u, v \left(u \rightarrow v \Rightarrow \neg(u \in U_{\spadesuit} \wedge v \in U_{\spadesuit}) \wedge \right. \\ & \quad \quad \left. \neg(u \in U_{\heartsuit} \wedge v \in U_{\heartsuit}) \wedge \neg(u \in U_{\clubsuit} \wedge v \in U_{\clubsuit}) \right) \end{aligned}$$

The existentially quantified set variables U_{\spadesuit} , U_{\heartsuit} and U_{\clubsuit} represent the three possible colors. In the first two lines, we specify that the sets assigned to these variables form a partition of the set of nodes (possibly with empty components). The remaining two lines constitute the actual definition of a valid coloring: no two adjacent nodes share the same color, which means that adjacent nodes are in different sets.

A *first-order formula* (FO-formula) is an MSO-formula in which set variables may not be bound by quantifiers, i.e., subformulas of the form $\exists X(\varphi)$ and $\forall X(\varphi)$ are disallowed, for $X \in \mathcal{V}_{\text{set}}$. An *existential MSO-formula* (EMSO-formula) is of the form $\exists X_1, \dots, X_n(\varphi)$, where $X_1, \dots, X_n \in \mathcal{V}_{\text{set}}$ and φ is an FO-formula. We denote the classes of FO- and EMSO-definable graph languages by \mathcal{L}_{FO} and $\mathcal{L}_{\text{EMSO}}$. (Note that by Example 2.2.2, the language of 3-colorable graphs lies in $\mathcal{L}_{\text{EMSO}}$.)

3 Distributed Graph Automata

The simple idea of interconnecting finite-state machines in a synchronous distributed setting presents a natural paradigm for defining finite automata on graphs of arbitrary topology. In this section, we introduce three classes of automata obtained this way, and discuss some of their properties. Our most powerful version of distributed graph automata turns out to be equivalent to MSO-logic on graphs. The other two are restricted variants for which the emptiness problem is decidable.

3.1 Informal Description

We start with an informal description of our automaton model. Formal definitions follow in subsection 3.2.

A distributed graph automaton (DGA) is an abstract machine that, given a labeled graph as input, can either accept or reject it, thereby specifying a graph language. Our model of computation incorporates the following key concepts:

Synchronous Distributed Algorithm. A DGA operates primarily as a distributed algorithm. Each node of the input graph is assigned its own local processor, which we shall not distinguish from the node itself. Communication takes place in synchronous rounds, in which each node receives the current states of its incoming neighbors.

Finite-State Machines. Each local processor is a finite-state machine, i.e., an abstract machine that can be in one of a finite number of states, and has no additional memory. Its initial state is determined by the node label. After each communication round, it updates its state according to a (possibly nondeterministic) transition function that depends only on the current state and the states received from the incoming neighborhood.

Constant Running Time. The number of communication rounds is limited by a constant. To ensure this, we associate a number, called *level*, with every state. In most cases, this number indicates the round in which the state may occur. We require that potentially initial states are at level 0, and outgoing transitions from states at level i go to states at level $i + 1$. There is an exception, however: the states at the highest level, called the *permanent states*, can also be initial states, and can have incoming transitions from any level. Moreover, all their outgoing transitions are self-loops. The idea is that, once a node has reached a permanent state, it terminates its local computation, and waits for the other nodes in the graph to terminate too.

Aggregation of States. In order to be finitely representable, a DGA treats collections of states as sets, i.e., it abstracts away from the multiplicity of states. This aggregation of states into sets is applied in two ways:

- First, the information received by the nodes in each round is a family of sets of states, indexed by the edge alphabet of the graph. That is, for each edge relation, a node knows which states occur in its incoming neighborhood, but it cannot distinguish between neighbors that are in the same state.
- Second, once all the nodes have reached a permanent state, the DGA ceases to operate as a distributed algorithm, and collects all the reached permanent states into a set F . This set is the sole acceptance criterion: if F is part of the DGA's accepting sets, then the input graph is accepted, otherwise it is rejected.

As an introductory example, let us translate the MSO-formula φ_3^{color} from Example 2.2.2 to the setting of DGAs.

3.1.1 Example (3-Colorability).

Figure 1 shows the state diagram of a simple nondeterministic DGA $\mathcal{A}_3^{\text{color}}$. The states are arranged in columns corresponding to their levels, ascending from left to right. $\mathcal{A}_3^{\text{color}}$ expects a $\{\square\}$ -labeled $\{\square\}$ -graph as input, and accepts it if and only if it is 3-colorable. The automaton proceeds as follows: All nodes of the input graph are initialized to the state q_{ini} . In the first round, each node nondeterministically chooses to go to one of the states q_{\spadesuit} , q_{\heartsuit} and q_{\clubsuit} , which represent the three possible colors. Then, in the second round, the nodes verify locally that the chosen coloring is valid. If the set received from their incoming neighborhood (only one, since there is only a single edge relation) contains their own state, they go to q_{no} , otherwise to q_{yes} . The automaton then accepts the input graph if and only if all the nodes are in q_{yes} , i.e., $\{q_{\text{yes}}\}$ is its only accepting set. This is indicated by the blue bar to the right of the state diagram. We shall refer to such a representation of sets using bars as *barcode*.

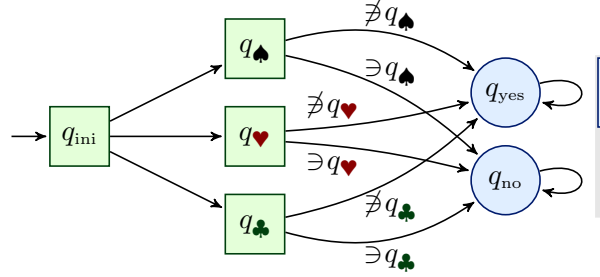


Figure 1. $\mathcal{A}_3^{\text{color}}$, a nondeterministic DGA over $\langle\{\square\}, \{\square\}\rangle$ whose graph language consists of the 3-colorable graphs.

One last key concept that enters into our most general definition of DGAs is *alternation*, a generalization of nondeterminism introduced by Chandra, Kozen and Stockmeyer in [CKS81] (in their case, for Turing machines and other types of word automata).

Alternating Automaton. In addition to being able to nondeterministically choose between different transitions, nodes can also explore several choices in parallel. To this end, the nonpermanent states of an alternating DGA (ADGA) are partitioned into two types, *existential* and *universal*, such that states on the same level are of the same type. If, in a given round, the nodes are in existential states, then they nondeterministically choose a single state to go to in the next round, as described above. In contrast, if they are in universal states, then the run of the ADGA is split into several parallel branches, called universal branches, one for each possible combination of choices of the nodes. This procedure of splitting is repeated recursively for each round in which the nodes are in universal states. The ADGA then accepts the input graph if and only if its acceptance condition is satisfied in every universal branch of the run.

3.1.2 Example (Non-3-Colorability).

To illustrate the notion of universal branching, consider the ADGA $\bar{\mathcal{A}}_3^{\text{color}}$ shown in Fig. 2. It is a complement automaton of $\mathcal{A}_3^{\text{color}}$ from Example 3.1.1, i.e., it accepts precisely those $\{\square\}$ -labeled $\{\square\}$ -graphs that are *not* 3-colorable. States represented as red triangles are

universal (whereas the green squares in Fig. 1 stand for existential states). Given an input graph with n nodes, $\bar{\mathcal{A}}_3^{\text{color}}$ proceeds as follows: All nodes are initialized to q_{ini} . In the first round, the run is split into 3^n universal branches, each of which corresponds to one possible outcome of the first round of $\mathcal{A}_3^{\text{color}}$ running on the same input graph. Then, in the second round, in each of the 3^n universal branches, the nodes check whether the coloring chosen in that branch is valid. As indicated by the barcode, the acceptance condition of $\bar{\mathcal{A}}_3^{\text{color}}$ is satisfied if and only if at least one node is in state q_{no} , i.e., the accepting sets are $\{q_{\text{no}}\}$ and $\{q_{\text{yes}}, q_{\text{no}}\}$. Hence, the automaton accepts the input graph if and only if no valid coloring was found in any universal branch. Note that we could also have chosen to make the states q_{\spadesuit} , q_{\heartsuit} and q_{\clubsuit} existential, since their outgoing transitions are deterministic. Regardless of their type, there is no branching in the second round.

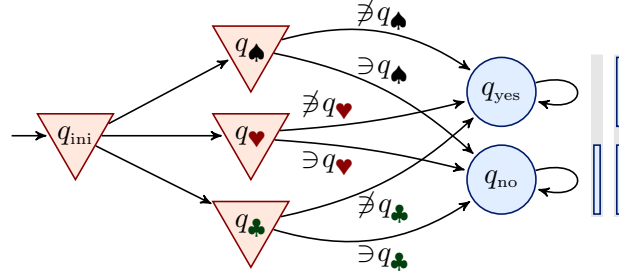


Figure 2. $\bar{\mathcal{A}}_3^{\text{color}}$, an alternating DGA over $\langle \{\square\}, \{\square\} \rangle$ whose graph language consists of the graphs that are *not* 3-colorable.

3.2 Formal Definitions

We now repeat and clarify the notions from subsection 3.1 in a more formal setting, beginning with our most general definition of DGAs.

3.2.1 Definition (Alternating Distributed Graph Automaton).

An *alternating distributed graph automaton* (ADGA) over alphabets $\langle \Sigma, \Gamma \rangle$ is a tuple $\mathcal{A} = \langle \Sigma, \Gamma, \hat{Q}, \sigma, \delta, \mathcal{F} \rangle$, where

- Σ and Γ are finite nonempty sets of node labels and edge labels, respectively,
- $\hat{Q} = \langle Q_{\exists}, Q_{\forall}, Q_{\text{P}} \rangle$, where Q_{\exists} , Q_{\forall} and Q_{P} , with $Q_{\text{P}} \neq \emptyset$, are pairwise disjoint finite sets of *existential*, *universal* and *permanent* states, respectively, which are also referred to by the notational shorthands
 - $Q := Q_{\exists} \cup Q_{\forall} \cup Q_{\text{P}}$, for the entire set of *states*,
 - $Q_{\text{N}} := Q_{\exists} \cup Q_{\forall}$, for the set of *nonpermanent* states,
- $\sigma: \Sigma \rightarrow Q$ is an *initialization function*,
- $\delta: Q \times (2^Q)^{\Gamma} \rightarrow 2^Q$ is a (local) *transition function*, and
- $\mathcal{F} \subseteq 2^{Q_{\text{P}}}$ is a set of *accepting sets* of permanent states.

The functions σ and δ must be such that one can unambiguously associate with every state $q \in Q$ a *level* $l_{\mathcal{A}}(q) \in \mathbb{N}$ satisfying the following conditions:

- States on the same level are of the same type, i.e., for every $i \in \mathbb{N}$,
$$\{q \in Q \mid l_{\mathcal{A}}(q) = i\} \in (2^{Q_{\exists}} \cup 2^{Q_{\forall}} \cup 2^{Q_{\text{P}}}).$$

- Initial states are either on the lowest level or permanent, i.e., for every $q \in Q$,
 $\exists a \in \Sigma: \sigma(a) = q \quad \text{implies} \quad l_{\mathcal{A}}(q) = 0 \vee q \in Q_P.$

- Nonpermanent states without incoming transitions are on the lowest level, and transitions between nonpermanent states go only from one level to the next, i.e., for every $q \in Q_N$,

$$l_{\mathcal{A}}(q) = \begin{cases} 0 & \text{if for all } p \in Q \text{ and } \hat{S} \in (2^Q)^\Gamma, \text{ it holds that } q \notin \delta(p, \hat{S}), \\ i + 1 & \text{if there are } p \in Q_N \text{ and } \hat{S} \in (2^Q)^\Gamma \text{ such that} \\ & l_{\mathcal{A}}(p) = i \text{ and } q \in \delta(p, \hat{S}). \end{cases}$$

- The permanent states are one level higher than the highest nonpermanent ones, and have only self-loops as outgoing transitions, i.e., for every $q \in Q_P$,

$$l_{\mathcal{A}}(q) = \begin{cases} 0 & \text{if } Q_N = \emptyset, \\ \max\{l_{\mathcal{A}}(q) \mid q \in Q_N\} + 1 & \text{otherwise,} \end{cases}$$

$$\delta(q, \hat{S}) = \{q\} \quad \text{for every } \hat{S} \in (2^Q)^\Gamma.$$

For any ADGA $\mathcal{A} = \langle \Sigma, \Gamma, \hat{Q}, \sigma, \delta, \mathcal{F} \rangle$, we define its *length* $\text{len}(\mathcal{A})$ to be its highest level, i.e., $\text{len}(\mathcal{A}) := \max\{l_{\mathcal{A}}(q) \mid q \in Q\}$.

Next, we want to give a formal definition of a run. For this, we need the notion of a configuration, which can be seen as the global state of an ADGA.

3.2.2 Definition (Configuration).

Consider an ADGA $\mathcal{A} = \langle \Sigma, \Gamma, \hat{Q}, \sigma, \delta, \mathcal{F} \rangle$. We call any Q -labeled Γ -graph $G_\kappa \in Q^{\hat{Q}}$ a *configuration* of \mathcal{A} on G . If every node in G_κ is labeled by a permanent state, we refer to G_κ as a *permanent* configuration. Otherwise, if G_κ is a nonpermanent configuration whose nodes are labeled exclusively by existential and (possibly) permanent states, we say that G_κ is an *existential* configuration. Analogously, G_κ is *universal* if it is nonpermanent and only labeled by universal and (possibly) permanent states.

Additionally, we say that a permanent configuration G_κ is *accepting* if the set of states occurring in it is accepting, i.e., if $\{\kappa(v) \mid v \in V_G\} \in \mathcal{F}$. Any other permanent configuration is called *rejecting*. Nonpermanent configurations are neither accepting nor rejecting.

The (local) transition function of an ADGA specifies for each state a set of potential successors, for a given family of sets of states. This can be naturally extended to configurations, which leads us to the definition of a global transition function.

3.2.3 Definition (Global Transition Function).

The *global transition function* $\delta^{\hat{Q}}$ of an ADGA $\mathcal{A} = \langle \Sigma, \Gamma, \hat{Q}, \sigma, \delta, \mathcal{F} \rangle$ assigns to each configuration G_κ of \mathcal{A} the set of all of its *successor configurations* G_μ , by combining all possible outcomes of local transitions on G_κ , i.e.,

$$\delta^{\hat{Q}}: Q^{\hat{Q}} \rightarrow 2^{(Q^{\hat{Q}})}$$

$$G_\kappa \mapsto \left\{ G_\mu \mid \bigwedge_{v \in V_G} \mu(v) \in \delta\left(\kappa(v), \langle \{\kappa(u) \mid u \xrightarrow{\gamma_G} v \rangle_{\gamma \in \Gamma}\right) \right\}.$$

We now have everything at hand to formalize the notion of a run.

3.2.4 Definition (Run).

A *run* of an ADGA $\mathcal{A} = \langle \Sigma, \Gamma, \widehat{Q}, \sigma, \delta, \mathcal{F} \rangle$ on a labeled graph $G_\lambda \in \Sigma^{\widehat{Q}}$ is a directed acyclic graph $R = \langle K, \rightarrow \rangle$ whose nodes are configurations of \mathcal{A} on G , such that

- the *initial configuration* $G_{\sigma \circ \lambda} \in K$ is the only source,¹
- every nonpermanent configuration $G_\kappa \in K$ with $\delta^{\widehat{Q}}(G_\kappa) = \{G_{\mu_1}, \dots, G_{\mu_m}\}$ has
 - exactly one outgoing neighbor $G_{\mu_i} \in \delta^{\widehat{Q}}(G_\kappa)$ if G_κ is existential,
 - exactly m outgoing neighbors $G_{\mu_1}, \dots, G_{\mu_m}$ if G_κ is universal, and
- every permanent configuration $G_\kappa \in K$ is a sink.

The run R is *accepting* if every permanent configuration $G_\kappa \in K$ is accepting.

An ADGA $\mathcal{A} = \langle \Sigma, \Gamma, \widehat{Q}, \sigma, \delta, \mathcal{F} \rangle$ *accepts* a labeled graph $G_\lambda \in \Sigma^{\widehat{Q}}$ if and only if there exists an accepting run R of \mathcal{A} on G_λ . The graph language *recognized* by \mathcal{A} is the set

$$\mathbf{L}(\mathcal{A}) := \{G_\lambda \in \Sigma^{\widehat{Q}} \mid \mathcal{A} \text{ accepts } G_\lambda\}.$$

Every graph language that is recognized by some ADGA is called *ADGA-recognizable*. We denote by $\mathcal{L}_{\text{ADGA}}$ the class of all such graph languages.

The ADGA \mathcal{A} is *equivalent* to some MSO(Σ, Γ)-sentence φ if it recognizes precisely the graph language defined by φ , i.e., if $\mathbf{L}(\mathcal{A}) = \mathbf{L}_{\Sigma, \Gamma}(\varphi)$.

We inductively define that a configuration $G_\kappa \in Q^{\widehat{Q}}$ is *reachable* by \mathcal{A} on G_λ if either $G_\kappa = G_{\sigma \circ \lambda}$, or $G_\kappa \in \delta^{\widehat{Q}}(G_\mu)$ for some configuration $G_\mu \in Q^{\widehat{Q}}$ reachable by \mathcal{A} on G_λ . In case G_λ is irrelevant, we simply say that G_κ is reachable by \mathcal{A} .

The automaton \mathcal{A} is called a *nondeterministic DGA* (NDGA) if it has no universal states, i.e., if $Q_V = \emptyset$. If additionally every configuration $G_\kappa \in Q^{\widehat{Q}}$ that is reachable by \mathcal{A} has precisely one successor configuration, i.e., $|\delta^{\widehat{Q}}(G_\kappa)| = 1$, then we refer to \mathcal{A} as a *deterministic DGA* (DDGA). We denote the classes of NDGA- and DDGA-recognizable graph languages by $\mathcal{L}_{\text{NDGA}}$ and $\mathcal{L}_{\text{DDGA}}$.

Let us now illustrate the notion of ADGA by means of a slightly more involved example.

3.2.5 Example (Concentric Circles).

Consider the ADGA $\mathcal{A}_{\text{centric}} = \langle \Sigma, \Gamma, \widehat{Q}, \sigma, \delta, \mathcal{F} \rangle$ represented by the state diagram in Fig. 3. The node and edge alphabets are $\Sigma = \{a, b, c\}$ and $\Gamma = \{\square\}$. Again, existential states are represented by green squares, universal states by red triangles, and permanent states by blue circles. The short arrows mapping node labels to states indicate the initialization function σ . For instance, $\sigma(a) = q_a$. The other arrows specify the transition function δ . A label on such a transition arrow indicates a requirement on the set of states that a node receives from its incoming neighborhood (only one set, since there is only a single edge relation). For instance, $\delta(q_b, \langle \{q_a, q_c\} \rangle) = \{q_{b\clubsuit}, q_{b\blacklozenge}\}$. If there is no label, any set is permitted. Finally, as indicated by the barcode on the far right, the set of accepting sets is $\mathcal{F} = \{\{q_{a\spadesuit}, q_{\text{yes}}\}, \{q_{a\heartsuit}, q_{\text{yes}}\}\}$.

Intuitively, $\mathcal{A}_{\text{centric}}$ proceeds as follows: In the first round, the *a*-labeled nodes do nothing but update their state, while the *b*- and *c*-labeled nodes verify that the labels in their incoming neighborhood satisfy the condition of a valid graph coloring. The *c*-labeled nodes

¹As before, the operator \circ denotes function composition, such that $(\sigma \circ \lambda)(v) = \sigma(\lambda(v))$.

additionally check that they do not see any a 's, and then directly terminate. Meanwhile, the b -labeled nodes nondeterministically choose one of the markers ♣ and ♦. In the second round, only the a -labeled nodes are busy. They verify that their incoming neighborhood consists exclusively of b -labeled nodes, and that both of the markers ♣ and ♦ are present, thus ensuring that they have at least two incoming neighbors. Then, they simultaneously pick the markers ♠ and ♥, thereby creating different universal branches, and the run of the automaton terminates. Finally, the ADGA checks that all the nodes approve of the graph (meaning that none of them has reached the state q_{no}), and that in each universal branch, precisely one of the markers ♠ and ♥ occurs, which implies that there is a unique a -labeled node.

To sum up, the graph language $L(\mathcal{A}_{\text{centric}})$ consists of all the $\{a, b, c\}$ -labeled $\{\square\}$ -graphs such that

- the labeling constitutes a valid 3-coloring,
- there is precisely one a -labeled node v_a , and
- v_a has only b -labeled nodes in its undirected neighborhood, and at least two incoming neighbors.

The name “ $\mathcal{A}_{\text{centric}}$ ” refers to the fact that, in the (weakly) connected component of v_a , the b - and c -labeled nodes form *concentric* circles around v_a , i.e., nodes at distance 1 of v_a are labeled with b , nodes at distance 2 (if existent) with c , nodes at distance 3 (if existent) with b , and so forth.

Figure 4 shows an example of a labeled graph that lies in $L(\mathcal{A}_{\text{centric}})$. A corresponding accepting run can be seen in Fig. 5. We have adopted the same coloring scheme as for (automaton) states, i.e., a green configuration is existential, a red one is universal, and a blue one is permanent. In the first round, the three nodes that are in state q_b have a nondeterministic choice between $q_{b\clubsuit}$ and $q_{b\diamond}$. Hence, the second configuration is one of eight possible choices. The branching in the second round is due to the node in state q'_a which goes simultaneously to $q_{a\spadesuit}$ and $q_{a\heartsuit}$. In both branches, an accepting configuration is reached, since $\{q_{a\spadesuit}, q_{\text{yes}}\}$ and $\{q_{a\heartsuit}, q_{\text{yes}}\}$ are both accepting sets. Therefore, the entire run is accepting.

In the following subsections (3.3, 3.4 and 3.5), we derive our results on the properties of DGAs. For more detailed proofs and further examples of automata (recognizing, e.g., connected or cyclic graphs), see [Rei14].

3.3 Hierarchy and Closure Properties

3.3.1 Lemma (Closure Properties of $\mathcal{L}_{\text{ADGA}}$).

The class $\mathcal{L}_{\text{ADGA}}$ of ADGA-recognizable graph languages is effectively closed under Boolean set operations and under projection.

Proof sketch. As usual for alternating automata, complementation can be achieved by simply swapping the existential and universal states, and complementing the acceptance condition. That is, for an ADGA $\mathcal{A} = \langle \Sigma, \Gamma, \langle Q_{\exists}, Q_{\forall}, Q_P \rangle, \sigma, \delta, \mathcal{F} \rangle$, a complement automaton is $\bar{\mathcal{A}} = \langle \Sigma, \Gamma, \langle Q_{\forall}, Q_{\exists}, Q_P \rangle, \sigma, \delta, 2^{Q_P} \setminus \mathcal{F} \rangle$. This can be easily seen by associating a two-player game with \mathcal{A} and any Σ -labeled Γ -graph G_{λ} . One player tries to come up with an accepting run of \mathcal{A} on G_{λ} , whereas the other player seeks to find a (path to a)

rejecting configuration in any run proposed by the adversary. The first player has a winning strategy if and only if \mathcal{A} accepts G_λ . (This game-theoretic characterization will be used and explained more extensively in the proof of Theorem 3.4.1.) From this perspective, the construction of $\bar{\mathcal{A}}$ corresponds to interchanging the roles and winning conditions of the two players.

For two ADGAs \mathcal{A}_1 and \mathcal{A}_2 , we can effectively construct an ADGA \mathcal{A}_\cup that recognizes $L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$ by taking advantage of nondeterminism. The approach is, in principle, very similar to the corresponding construction for nondeterministic finite automata on words. In the first round of \mathcal{A}_\cup , each node in the input graph nondeterministically and independently decides whether to behave like in \mathcal{A}_1 or in \mathcal{A}_2 . If there is a consensus, then the run continues as it would in the unanimously chosen automaton \mathcal{A}_j , and it is accepting if and only if it corresponds to an accepting run of \mathcal{A}_j . Otherwise, a conflict is detected, either locally by adjacent nodes that have chosen different automata, or at the latest, when acceptance is checked globally (important for disconnected graphs), and in either case the run is rejecting. (Note that we have omitted some technicalities that ensure that the construction outlined above satisfies all the properties of an ADGA.)

Closure under node projection is straightforward, again by exploiting nondeterminism. Given an ADGA \mathcal{A} with node alphabet Σ and a projection $h: \Sigma \rightarrow \Sigma'$, we can effectively construct an ADGA \mathcal{A}' that recognizes $h(L(\mathcal{A}))$ as follows: For every $b \in \Sigma'$, each node labeled with b nondeterministically chooses a new label $a \in \Sigma$, such that $h(a) = b$. Then, the automaton \mathcal{A} is simulated on that new input. ■

3.3.2 Lemma ($\mathcal{L}_{\text{NDGA}} \subset \mathcal{L}_{\text{ADGA}}$).

There are (infinitely many) ADGA-recognizable graph languages that are not NDGA-recognizable.

Proof. Let $\Sigma = \Gamma = \{\square\}$. For any constant $k \geq 1$, we consider the language $L_{\leq k}^{\text{order}}$ of all graphs that have at most k nodes, i.e., $L_{\leq k}^{\text{order}} = \{G \in \Sigma^\boxplus \mid |V_G| \leq k\}$. We can easily construct an ADGA that recognizes this graph language: In a universal branching, each node goes to $k+1$ different states in parallel. The automaton accepts if and only if there is no branch in which the $k+1$ states occur all at once. Now, assume for sake of contradiction that $L_{\leq k}^{\text{order}}$ is also recognized by some NDGA \mathcal{A} , and let G be a graph with k nodes. We construct a variant G' of G with $k+1$ nodes by duplicating some node v , together with all of its incoming and outgoing edges. Observe that any accepting run of \mathcal{A} on G can be extended to an accepting run on G' , where the copy of v behaves exactly like v in every round. ■

3.3.3 Lemma (Closure Properties of $\mathcal{L}_{\text{NDGA}}$).

The class $\mathcal{L}_{\text{NDGA}}$ of NDGA-recognizable graph languages is effectively closed under union, intersection and projection, but *not* closed under complementation.

Proof. For union and projection, we simply use the same constructions as for ADGAs (see Lemma 3.3.1).

Intersection can be handled by a product construction, similar to the one for finite automata on words. Given two NDGAs \mathcal{A}_1 and \mathcal{A}_2 , we construct an NDGA \mathcal{A}_\otimes that operates on the Cartesian product of the state sets of \mathcal{A}_1 and \mathcal{A}_2 . It simulates the two automata simultaneously and accepts if and only if both of them reach an accepting configuration.

To see that $\mathcal{L}_{\text{NDGA}}$ is not closed under complementation, we recall from the proof of Lemma 3.3.2 that for any $k \geq 1$, the language $L_{\leq k}^{\text{order}}$ of all graphs that have at most k nodes is not NDGA-recognizable. However, complementing the ADGA given for $L_{\leq k}^{\text{order}}$ yields an NDGA that recognizes the complement language $L_{\geq k+1}^{\text{order}}$. ■

3.3.4 Lemma ($\mathcal{L}_{\text{DDGA}} \subset \mathcal{L}_{\text{NDGA}}$).

There are (infinitely many) NDGA-recognizable graph languages that are not DDGA-recognizable.

Proof. Let $k \geq 2$. As mentioned in the proof of Lemma 3.3.3, the language $L_{\geq k}^{\text{order}}$ of all graphs that have at least k nodes is NDGA-recognizable. To see that it is not DDGA-recognizable, consider (similarly to the proof of Lemma 3.3.2) a graph G with $k - 1$ nodes and a variant G' with k nodes obtained from G by duplicating some node v , together with all of its incoming and outgoing edges. Given any DDGA \mathcal{A} , the determinism of \mathcal{A} guarantees that v and its copy v' behave the same way in the (unique) run of \mathcal{A} on G' . Hence, if that run is accepting, so is the run on G . ■

3.3.5 Lemma (Closure Properties of $\mathcal{L}_{\text{DDGA}}$).

The class $\mathcal{L}_{\text{DDGA}}$ of DDGA-recognizable graph languages is effectively closed under Boolean set operations, but *not* closed under projection.

Proof. To complement a DDGA, we can simply complement its set of accepting sets. The product construction for intersection of NDGAs mentioned in Lemma 3.3.3 remains applicable when restricted to DDGAs.

Closure under node projection does not hold because we can, for instance, construct a DDGA that recognizes the language $L_{a,b,c}^{\text{occur}}$ of all $\{a, b, c\}$ -labeled graphs in which each of the three node labels occurs at least once. However, projection under the mapping $h: \{a, b, c\} \rightarrow \{\square\}$, with $h(a) = h(b) = h(c) = \square$, yields the graph language $h(L_{a,b,c}^{\text{occur}}) = L_{\geq 3}^{\text{order}}$, which is not DDGA-recognizable (see the proof of Lemma 3.3.4). ■

3.4 Equivalence of ADGAs and MSO-Logic

3.4.1 Theorem ($\mathcal{L}_{\text{ADGA}} = \mathcal{L}_{\text{MSO}}$).

A graph language is ADGA-recognizable if and only if it is MSO-definable. There are effective translations in both directions.

Proof sketch.

(\Rightarrow) We start with the direction $\mathcal{L}_{\text{ADGA}} \subseteq \mathcal{L}_{\text{MSO}}$. Let $\mathcal{A} = \langle \Sigma, \Gamma, \widehat{Q}, \sigma, \delta, \mathcal{F} \rangle$ be an ADGA of length n . Without loss of generality, we may assume that every configuration reachable by \mathcal{A} has at least one successor configuration and that no permanent configuration is reachable in less than n rounds. In order to encode the acceptance behaviour of \mathcal{A} into an MSO(Σ, Γ)-sentence $\varphi_{\mathcal{A}}$, we take again the game-theoretic point of view² briefly mentioned in the proof sketch of Lemma 3.3.1. Given \mathcal{A} and some $G_{\lambda} \in \Sigma^{\widehat{V}}$, we consider a game with two players: the *automaton* (player \exists) and the *pathfinder* (player \forall). This game is represented by a directed acyclic graph

²This characterization is heavily inspired by the work of Löding and Thomas in [LT00].

whose nodes are precisely the configurations reachable by \mathcal{A} on G_λ . For any two *nonpermanent* configurations G_κ and G_μ , there is a directed edge from G_κ to G_μ if and only if $G_\mu \in \delta^\diamond(G_\kappa)$. Starting at the initial configuration $G_{\sigma \circ \lambda}$, the two players move through the game together by following directed edges. If the current configuration is existential, then the automaton has to choose the next move, if it is universal, then the decision belongs to the pathfinder. This continues until some permanent configuration is reached. The automaton wins if that permanent configuration is accepting, whereas the pathfinder wins if it is rejecting. A player is said to have a *winning strategy* if it can always win, independently of its opponent's moves. It is straightforward to prove that the automaton has a winning strategy if and only if \mathcal{A} accepts G_λ . Our MSO-sentence $\varphi_{\mathcal{A}}$ will express the existence of such a winning strategy, and thus be equivalent to \mathcal{A} .

Within MSO-logic, we represent a path $\pi = G_{\kappa_0} \cdots G_{\kappa_n}$ through the game by a sequence of families of set variables $\hat{X}_0, \dots, \hat{X}_n$, where $\hat{X}_0 = \langle \rangle$ and $\hat{X}_i = \langle \mathbf{U}_{i,q} \rangle_{q \in Q}$, for $1 \leq i \leq n$. The intention is that each set variable $\mathbf{U}_{i,q}$ is interpreted as the set of nodes $v \in V_G$ for which $\kappa_i(v) = q$. (We do not need set variables to represent G_{κ_0} , since the players always start at $G_{\sigma \circ \lambda}$.)

Now, for every round i , we construct a formula $\varphi_i^{\text{win}}[\hat{X}_i]$ (i.e., with free variables in \hat{X}_i), which expresses that the automaton has a winning strategy in the subgame starting at the configuration G_{κ_i} represented by \hat{X}_i . In case G_{κ_i} is existential, this is true if the automaton has a winning strategy in some successor configuration of G_{κ_i} , whereas if G_{κ_i} is universal, the automaton must have a winning strategy in all successor configurations of G_{κ_i} . This yields the following recursive definition for $0 \leq i \leq n-1$:

$$\varphi_i^{\text{win}}[\hat{X}_i] := \begin{cases} \exists \hat{X}_{i+1} \left(\varphi_{i+1}^{\text{succ}}[\hat{X}_i, \hat{X}_{i+1}] \wedge \varphi_{i+1}^{\text{win}}[\hat{X}_{i+1}] \right) & \text{if level } i \text{ of } \mathcal{A} \\ & \text{is existential,} \\ \forall \hat{X}_{i+1} \left(\varphi_{i+1}^{\text{succ}}[\hat{X}_i, \hat{X}_{i+1}] \Rightarrow \varphi_{i+1}^{\text{win}}[\hat{X}_{i+1}] \right) & \text{if level } i \text{ of } \mathcal{A} \\ & \text{is universal.} \end{cases}$$

Here, $\varphi_{i+1}^{\text{succ}}[\hat{X}_i, \hat{X}_{i+1}]$ is an FO-formula expressing that \hat{X}_i and \hat{X}_{i+1} represent two configurations G_{κ_i} and $G_{\kappa_{i+1}}$ such that $G_{\kappa_{i+1}} \in \delta^\diamond(G_{\kappa_i})$. As our recursion base, we can easily construct a formula $\varphi_n^{\text{win}}[\hat{X}_n]$ that is satisfied if and only if \hat{X}_n represents an accepting configuration of \mathcal{A} .

The desired MSO-sentence is $\varphi_{\mathcal{A}} := \varphi_0^{\text{win}}[\hat{X}_0] = \varphi_0^{\text{win}}[\]$.

- (\Leftarrow) For the direction $\mathcal{L}_{\text{ADGA}} \supseteq \mathcal{L}_{\text{MSO}}$, we can proceed by induction on the structure of an MSO(Σ, Γ)-formula φ . In order to deal with free occurrences of variables, we encode variable assignments into node labels. For $G_\lambda \in \Sigma^{\hat{\mathcal{U}}}$ and $\alpha: \text{free}(\varphi) \rightarrow V_G \cup 2^{V_G}$, we represent $\langle G_\lambda, \alpha \rangle$ as the labeled graph $G_{\lambda \times \alpha^{-1}}$ whose labeling $\lambda \times \alpha^{-1}$ assigns to each node $v \in V_G$ the tuple $\langle \lambda(v), \alpha^{-1}(v) \rangle$, where $\alpha^{-1}(v)$ is the set of all variables in $\text{free}(\varphi)$ to which α assigns either v or a set containing v . We now inductively construct an ADGA $\mathcal{A}_\varphi = \langle \Sigma \times 2^{\text{free}(\varphi)}, \Gamma, \hat{Q}, \sigma, \delta, \mathcal{F} \rangle$ such that

$$G_{\lambda \times \alpha^{-1}} \in L(\mathcal{A}_\varphi) \quad \text{if and only if} \quad \langle G_\lambda, \alpha \rangle \models \varphi.$$

(BC) Let $b \in \Sigma$, $\tau \in \Gamma$, $x, y \in \mathcal{V}_{\text{node}}$ and $X \in \mathcal{V}_{\text{set}}$.

If φ is one of the atomic formulas $\Diamond x$, $x = y$ or $x \in X$, then, in \mathcal{A}_φ , each node simply checks that its own label $\langle a, M \rangle \in \Sigma \times 2^{\text{free}(\varphi)}$ satisfies the condition specified in φ (which, in particular, is the case if $x, y \notin M$). Since this can be directly encoded into the initialization function σ , the ADGA has length 0. It accepts the input graph if and only if every node reports that its label satisfies the condition.

The case $\varphi = x \xrightarrow{\tau} y$ is very similar, but \mathcal{A}_φ needs one communication round, after which the node assigned to y can check whether it has received a message through a τ -edge from the node assigned to x . Accordingly, \mathcal{A}_φ has length 1.

(IS) In case φ is a composed formula, we can obtain \mathcal{A}_φ by means of the constructions outlined in the proof sketch of Lemma 3.3.1 (closure properties of $\mathcal{L}_{\text{ADGA}}$). Let ψ and ψ' be $\text{MSO}(\Sigma, \Gamma)$ -formulas with equivalent ADGAs \mathcal{A}_ψ and $\mathcal{A}_{\psi'}$, respectively.

If $\varphi = \neg\psi$, it suffices to define $\mathcal{A}_\varphi = \bar{\mathcal{A}}_\psi$. Similarly, if $\varphi = \psi \vee \psi'$, we get \mathcal{A}_φ by applying the union construction on \mathcal{A}_ψ and $\mathcal{A}_{\psi'}$. (In general, we first have to extend \mathcal{A}_ψ and $\mathcal{A}_{\psi'}$ such that they both operate on the same node alphabet $\Sigma \times 2^{\text{free}(\psi) \cup \text{free}(\psi')}$.)

Existential quantification can be handled by node projection. If $\varphi = \exists X(\psi)$, with $X \in \mathcal{V}_{\text{set}}$, we construct \mathcal{A}_φ by applying the projection construction on \mathcal{A}_ψ , using the mapping $h: \Sigma \times 2^{\text{free}(\psi)} \rightarrow \Sigma \times 2^{\text{free}(\varphi) \setminus \{X\}}$ that deletes the set variable X from every label. An analogous approach can be used if $\varphi = \exists x(\psi)$, with $x \in \mathcal{V}_{\text{node}}$. The only difference is that, instead of applying the projection construction directly on \mathcal{A}_ψ , we apply it on a variant \mathcal{A}'_ψ that operates just like \mathcal{A}_ψ , but additionally checks that precisely one node in the input graph is assigned to the variable x . ■

From Theorem 3.4.1 we can immediately infer that it is undecidable whether the graph language recognized by some arbitrary ADGA is empty. Otherwise, we could decide the satisfiability problem of MSO-logic on graphs, which is known to be undecidable (a direct consequence of Trakhtenbrot's Theorem, see, e.g., [Lib04, Thm 9.2]).

3.4.2 Corollary (Emptiness Problem of ADGAs).

The emptiness problem of ADGAs is undecidable.

3.5 Emptiness Problem of NDGAs

At the cost of reduced expressive power, we can also obtain a positive decidability result.

3.5.1 Lemma (Emptiness Problem of NDGAs).

The emptiness problem of NDGAs is decidable in doubly-exponential time. More precisely, for every NDGA $\mathcal{A} = \langle \Sigma, \Gamma, \widehat{Q}, \sigma, \delta, \mathcal{F} \rangle$, whether its recognized graph language $L(\mathcal{A})$ is empty or not can be decided in time 2^k , where $k \in O(|\Gamma| \cdot |Q|^{4 \cdot \text{len}(\mathcal{A})} \cdot \text{len}(\mathcal{A}))$.

Furthermore, whether or not $L(\mathcal{A})$ contains any *connected*, *undirected* graph can be decided in time $2^{2^{k'}}$, where $k' \in O(|\Gamma| \cdot |Q| \cdot \text{len}(\mathcal{A}))$.

Proof sketch. Let $G_\lambda \in \Sigma^{\widehat{T}}$. Since NDGAs cannot perform universal branching, we can consider any run of \mathcal{A} on G_λ as a sequence of configurations $R = G_{\kappa_0} \cdots G_{\kappa_n}$, with

$n \leq \text{len}(\mathcal{A})$. In R , each node of G traverses one of at most $|Q|^{\text{len}(\mathcal{A})+1}$ possible sequences of states. Now, assume that G has more than $|Q|^{\text{len}(\mathcal{A})+1}$ nodes. Then, by the Pigeonhole Principle, there must be two distinct nodes $v, v' \in V_G$ that traverse the same sequence of states in R . We construct a smaller graph G' by removing v' from G , together with its adjacent edges, and adding directed edges from v to all of the former *outgoing* neighbors of v' . If all the nodes in G' maintain their nondeterministic choices from R , none of them will notice that v' is missing, and consequently they all behave just as in R . The resulting run R' on G' is accepting if and only if R is accepting.

Applying this argument recursively, we conclude that if $L(\mathcal{A})$ is not empty, then it must contain some labeled graph that has at most $|Q|^{\text{len}(\mathcal{A})+1}$ nodes. Hence, the emptiness problem is decidable because the search space is finite. The time complexity indicated above corresponds to the naive approach of checking every (directed) graph with at most $|Q|^{\text{len}(\mathcal{A})+1}$ nodes.

If we are only interested in (connected) undirected graphs, the reasoning is very similar, but we have to require a larger minimum number of nodes in order to be able to remove some node without influencing the behavior of the others. In a graph G with more than $(|Q| \cdot 2^{|I| \cdot |Q|})^{\text{len}(\mathcal{A})+1}$ nodes, there must be two distinct nodes $v, v' \in V_G$ that, in addition to traversing the same sequence of states, also receive the same family of sets of states from their neighborhood in every round. Observe that the automaton will not notice if we merge v and v' . The rest of the argument is analogous to the previous scenario. ■

3.6 Summary and Discussion

We have introduced ADGAs, which are probably the first graph automata in the literature to be equivalent to MSO-logic on graphs. However, their expressive power results mainly from the use of alternation: we have seen that the deterministic, nondeterministic and alternating variants form a strict hierarchy, i.e.,

$$\mathcal{L}_{\text{DDGA}} \subset \mathcal{L}_{\text{NDGA}} \subset \mathcal{L}_{\text{ADGA}}.$$

The corresponding closure and decidability properties are summarized in Table 1.

	Closure Properties				Decidability
	Complement	Union	Intersection	Projection	Emptiness
ADGA	✓	✓	✓	✓	✗
NDGA	✗	✓	✓	✓	✓
DDGA	✓	✓	✓	✗	✓

Table 1. Closure and decidability properties of alternating, nondeterministic, and deterministic DGAs.

On an intuitive level, this hierarchy and these closure properties do not seem very surprising. One might even ask: *are ADGAs just another syntax for MSO-logic?* Indeed, universal branchings correspond to universal quantification, and nondeterministic choices to existential quantification. By disallowing universal set quantification in MSO-logic we obtain EMSO-logic, and further disallowing existential set quantification yields FO-logic. Analogously to DGAs, the classes of graph languages definable in these logics form a strict hierarchy, i.e.,

$$\mathcal{L}_{\text{FO}} \subset \mathcal{L}_{\text{EMSO}} \subset \mathcal{L}_{\text{MSO}}.$$

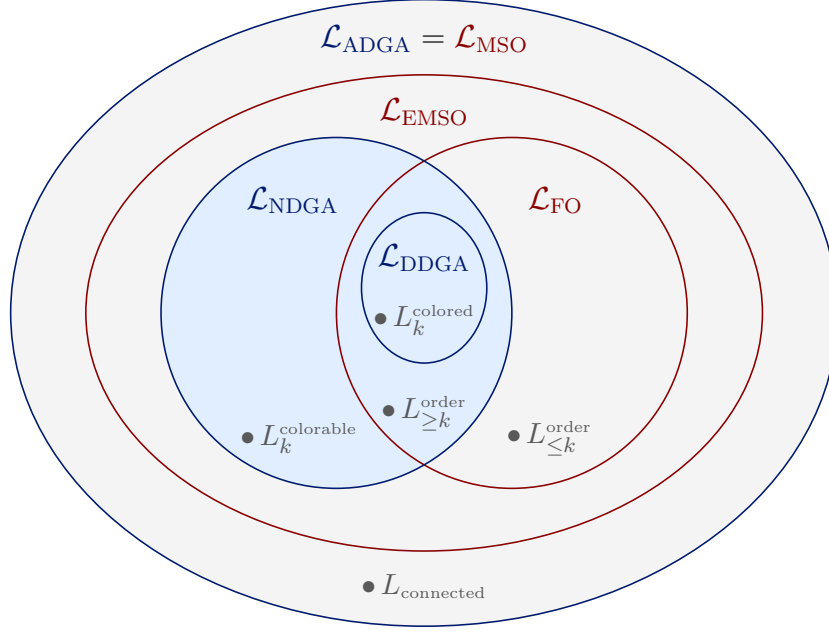


Figure 6. Venn diagram relating the classes of graph languages recognizable by our three flavors of DGAs to those definable in MSO-, EMSO- and FO-logic.

Furthermore, the closure properties of $\mathcal{L}_{\text{EMSO}}$ and \mathcal{L}_{FO} coincide with those of $\mathcal{L}_{\text{NDGA}}$ and $\mathcal{L}_{\text{DDGA}}$, respectively. Given that $\mathcal{L}_{\text{ADGA}}$ and \mathcal{L}_{MSO} are equal, one might therefore expect that the analogous equalities hold for the weaker classes. However, as already hinted by the positive decidability properties in Table 1, this is not the case. The actual relationships between the different classes of graph languages are depicted in Fig. 6. A glance at this Venn diagram suggests that ADGAs are not simply a one-to-one reproduction of MSO-logic.

Justification of Fig. 6. Fagin has shown in [Fag75] that the language $L_{\text{connected}}$ of all (weakly) connected graphs separates $\mathcal{L}_{\text{EMSO}}$ from \mathcal{L}_{MSO} . (Since non-connectivity is EMSO-definable, this also implies that $\mathcal{L}_{\text{EMSO}}$ is not closed under complementation.) The inclusion $\mathcal{L}_{\text{NDGA}} \subseteq \mathcal{L}_{\text{EMSO}}$ holds because we can encode every NDGA into an EMSO-sentence, using the same construction as in the proof sketch of Theorem 3.4.1. It is also easy to see that we do not need any set quantifiers to encode DDGAs, hence $\mathcal{L}_{\text{DDGA}} \subseteq \mathcal{L}_{\text{FO}}$. In the following, let $k, k' \geq 2$. The incomparability of $\mathcal{L}_{\text{NDGA}}$ and \mathcal{L}_{FO} is witnessed by the language $L_k^{\text{colorable}}$ of k -colorable graphs, which lies within $\mathcal{L}_{\text{NDGA}}$ (see Example 3.1.1) but outside of \mathcal{L}_{FO} (see, e.g., [Lib04]), and the language $L_{\leq k}^{\text{order}}$ of graphs with at most k nodes, which lies outside of $\mathcal{L}_{\text{NDGA}}$ (see the proof of Lemma 3.3.2) but obviously within \mathcal{L}_{FO} . Considering the union language $L_k^{\text{colorable}} \cup L_{\leq k'}^{\text{order}}$ also tells us that the inclusion of $\mathcal{L}_{\text{NDGA}} \cup \mathcal{L}_{\text{FO}}$ in $\mathcal{L}_{\text{EMSO}}$ is strict. Finally, the language $L_{\geq k}^{\text{order}}$ of graphs with at least k nodes separates $\mathcal{L}_{\text{DDGA}}$ from $\mathcal{L}_{\text{NDGA}} \cap \mathcal{L}_{\text{FO}}$ (see the proof of Lemma 3.3.4). A simple example of a language that lies within $\mathcal{L}_{\text{DDGA}}$ is the set L_k^{colored} of Σ -labeled graphs whose labelings are valid k -colorings, with $|\Sigma| = k$. ■

As of the time of writing this paper, no new results on \mathcal{L}_{MSO} have been inferred from the alternative characterization through ADGAs. On the other hand, the notion of NDGA contributes to the general observation, mentioned in Section 1, that many characterizations of regularity, which are equivalent on words and trees, drift apart on graphs. To see this, consider NDGAs whose input is restricted to those Σ -labeled Γ -graphs that represent words or trees over the alphabet Σ . For words, $\Gamma = \{\square\}$ and edges simply go from one

position to the next, whereas for ordered trees of arity k , we set $\Gamma = \{1, \dots, k\}$ and require edge relations such that $u \xrightarrow{i} v$ if and only if u is the i -th child of v . Observe that we can easily simulate any word or tree automaton by an NDGA of length 2: guess a run of the automaton in the first round (each node nondeterministically chooses some state), then check whether it is a valid accepting run in the second round (transitions are verified locally, and acceptance is determined by the unique sink). This implies that the classes of NDGA-recognizable and MSO-definable languages collapse on words and trees, and hence that NDGAs recognize precisely the regular languages on those restricted structures.

The fact that the emptiness problem of NDGAs is decidable on graphs seems noteworthy for several reasons:

- It can be seen as an extension to graphs of the corresponding decidability results for finite automata on words and trees, since, by the above remark, the emptiness problems of these automata correspond precisely to those of NDGAs restricted to words and trees, respectively.
- It might lead to the discovery of new decidable logics on graphs: a logic effectively equivalent to NDGAs would have a decidable satisfiability problem, and a logic effectively equivalent to DDGAs would additionally have a decidable validity problem. This could be interesting when contrasted with Trakhtenbrot’s Theorem, which states that these problems are undecidable for FO-logic, and a fortiori for (E)MSO-logic (see, e.g., [Lib04, Thm 9.2]).
- It implies that the language inclusion problem of DDGAs is also decidable: given two DDGAs \mathcal{A}_1 and \mathcal{A}_2 , we can decide whether $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$ by first applying the intersection construction on \mathcal{A}_1 and a complement of \mathcal{A}_2 , and then deciding emptiness for the resulting automaton. (This does not extend to NDGAs, since they do not satisfy closure under complementation.) The verification method presented in the next section is based on such an inclusion test.

4 Verification of Distributed Algorithms

The notion of graph automaton might have an application in formal verification of synchronous distributed algorithms. In this section, we consider a very simple toy example of such an algorithm, and suggest a mechanical verification technique based on DDGAs for proving partial correctness, using Floyd-Hoare logic. So far, our approach only works for an extremely restricted class of synchronous algorithms. However, since the method does not intrinsically depend upon a particular automaton model, it is possible, in principle, to extend it by replacing DDGAs with a more powerful class of graph automata. In this regard, the following method should be considered as an illustration of a concept, rather than a “ready-to-use” solution.

4.1 Distributed Programming Language

As mentioned by Konnov et al. in [KVW12], one of the major obstacles in formal verification of distributed algorithms is the lack of a versatile formal language to specify such algorithms. They refer to it as the *formalization problem*. Indeed, most of the distributed algorithms found in the literature are given as pseudocode, since implementation details are generally not the main concern.

Here, we restrict ourselves to a very weak class of synchronous algorithms for which the formalization problem can be easily solved. (This is not, by any means, an attempt at

a general solution.) We design our programming language in such a way that individual synchronous rounds can be simulated by a DDGA. In particular, this means that we only consider algorithms where

- the nodes have a finite state space,
- they send the same message to all of their neighbors, and
- they only receive a set containing all the messages sent by their neighbors.

Furthermore, in contrast to classical distributed algorithms, we express loops from the global point of view of a controller that can see the states of all the nodes at once. This will allow us to partly reason about distributed algorithms as if they were ordinary sequential algorithms and employ the inference rules from Floyd-Hoare logic. Obviously, the presence of a global controller introduces some additional expressive power which is not available in a purely distributed setting. We shall make use of it to model supplementary knowledge that the nodes might have about the graph. As a matter of fact, it is often assumed in distributed computing that the nodes know properties such as the order of the graph or its diameter.

We shall assume that our algorithms always run on *connected, undirected* graphs. The former property is usually required in distributed computing because nodes in separate connected components are unable to communicate with each other, which de facto means that any distributed algorithm is executed separately in each connected component. Assuming that graphs are undirected is also very common, and generally leads to simpler algorithms. In order to restrict the possible input graphs of our automata accordingly, for every DDGA \mathcal{A} , we denote by $L_*(\mathcal{A})$ the set of connected, undirected labeled graphs that are accepted by \mathcal{A} . For the remainder of this section, since no confusion with $L(\mathcal{A})$ can arise, we will also refer to $L_*(\mathcal{A})$ as the graph language *recognized* by \mathcal{A} , and say that it is *DDGA-recognizable*.

We now semi-formally specify the syntax and semantics of our distributed programming language. Any considered distributed algorithm operates on a set of *variables* $Var = \{x_1, \dots, x_k\}$ ranging over *values* from some *finite* domain Val . Each node v of the input graph has its own private copies of these variables, which are denoted by $v.x_1, \dots, v.x_k$ and are referred to as v 's *member variables*. The *global state* of a graph is given by a valuation of the member variables of all of its nodes. Formally, any (Val^{Var}) -labeled graph G_λ is a global state of the graph G , i.e., we label the nodes of G with functions from Var to Val .

The commands executed locally by a node v can contain expressions evaluated over Val . The syntax of these expressions is given by

$$e ::= v.x \mid f(e, \dots, e) \mid f(M, e, \dots, e),$$

where $x \in Var$, M is a special set variable that does not contribute to the global state, and f represents some function from $(Val \times \dots \times Val)$ or $(2^{Val} \times Val \times \dots \times Val)$ into Val .

Similarly, we allow Boolean expressions of the form

$$b ::= f(e, \dots, e) \mid f(M, e, \dots, e),$$

where the function associated with f maps into the Boolean domain.

As elementary *local commands*, any node v can either do nothing (**skip**) or assign a new value to one of its member variables. Furthermore, local commands can be composed sequentially and executed conditionally. The corresponding syntax is given by

$$C ::= \text{skip}; \mid v.x \leftarrow e; \mid C \ C \mid \text{if } b \text{ then } C \text{ else } C \text{ end.}$$

A *local command block* executed by v consists of a sequence of local commands. It can optionally be preceded by a synchronous message exchange, where v sends the value of one of its member variables $v.x$ to all of its neighbors, and in return receives a set of values which is assigned to the dedicated set variable M . The only purpose of M is to access the set of incoming messages, and its scope is restricted to the local command block.

$$D ::= C \mid \text{send } v.x \text{ receive } M; C$$

Next, we switch to a global perspective where we can control which local command blocks are executed by the nodes. As an elementary *global command*, we can tell all the nodes to execute a particular local command block synchronously in parallel. This corresponds to a single synchronous round of a distributed algorithm. To express more complex algorithms, global commands can be composed sequentially and executed in loops. The syntax is of the form

$$K ::= \text{each } v \text{ does } D \text{ end} \mid K K \mid \text{while } \xi \text{ do } K \text{ end},$$

where ξ is a textual representation of a condition on the global state of the graph that can be expressed as a DDGA-recognizable graph language. We compactly represent such conditions by FO-formulas over the node alphabet Val^{Var} , and refer to them as *DDGA-recognizable assertions*. Since DDGA-recognizable graph languages are closed under Boolean set operations, there are no restrictions on combining these formulas using the usual propositional connectives. Also, for convenience, we shall use many syntactic abbreviations whose meaning should be clear. For instance, by the abbreviation $\bigvee x_i = c$ we mean the disjunction of all the formulas $\Diamond a$ such that $a \in Val^{Var}$ and $a(x_i) = c$.

Let us consider the FLOODMAX algorithm as an example of a simple distributed algorithm that can be expressed in the programming language we just defined.

4.1.1 Example (FLOODMAX Algorithm).

Initially, each node is given a number from some finite domain. The task for the nodes is to compute the maximum number m present in the graph in such a way that, once the algorithm has terminated, all of them know m . An algorithm solving this problem can be used, for instance, to solve the leader election problem (see, e.g., [Lyn96, Sec 4.1]).

If we assume that the nodes know the diameter d of the graph, a simple approach is as follows: In each synchronous round, each node sends the maximum number it has seen so far (initially its own) to all of its neighbors. After d rounds, we are guaranteed that every node has received the global maximum.

In order to formalize this algorithm in our framework, we must somehow exploit the power of the global controller to simulate the circumstance that the nodes know d . However, since the controller can only check DDGA-recognizable assertions, it has no way of knowing d itself. Fortunately, this is not necessary. Counting the number of rounds up to d is only a way of ensuring that enough time has passed for information to propagate between any two nodes. Alternatively, the algorithm could also terminate as soon as no node receives any new information. Although this condition cannot be checked in a purely distributed setting, it can be specified by a DDGA, and thus leads to a variant of the algorithm that we can formalize.

A possible way of formalizing FLOODMAX can be seen in Algorithm 1. This algorithm operates on the variables m and m_{old} , which take values from some finite set I of nonnegative integers that contains 0. Initially, for each node v , a number in I is assigned to $v.m$

as input. Then, in each round, v updates its member variables in such a way that $v.m$ holds the maximum value it has seen so far and $v.m_{\text{old}}$ holds the value of $v.m$ from the previous round (except for the first round, where it is set to 0). The algorithm terminates as soon as for every node v the member variables $v.m$ and $v.m_{\text{old}}$ have the same value, a property (here represented by the assertion $\exists v(v.m \neq v.m_{\text{old}})$) that can be easily checked by a DDGA over the node alphabet $I^{\{m, m_{\text{old}}\}}$.

Algorithm 1. FLOODMAX

```

1 each  $v$  does
2   |  $v.m_{\text{old}} \leftarrow 0$ ;
3 end
4 while  $\exists v(v.m \neq v.m_{\text{old}})$  do
5   | each  $v$  does
6     | send  $v.m$  receive  $M$ ;
7     |  $v.m_{\text{old}} \leftarrow v.m$ ;
8     |  $v.m \leftarrow \max(M \cup \{v.m\})$ ;
9   | end
10 end

```

4.2 Verification Method

Now that we have a formal language for representing certain distributed algorithms, we can turn towards the verification method mentioned earlier. The basic idea is to consider a synchronous distributed algorithm as an ordinary sequential one, and treat each round of that algorithm as an atomic operation on the global state of the graph. Consequently, once we know how to derive a Hoare triple for a single round, we can simply use classical Floyd-Hoare logic to prove partial correctness of an entire algorithm.

In our framework, a synchronous round is represented by a global command $K = \text{each } v \text{ does } D \text{ end}$, where D is some local command block. We desire an inference rule that allows us to derive a Hoare triple of the form $\{\varphi\} K \{\psi\}$, where φ and ψ are DDGA-recognizable assertions on the global state of the graph.

As hinted previously, we can now take advantage of the restrictions that we have put on the considered algorithms in order to simulate K by a DDGA. This allows us to construct an automaton for the *weakest precondition* under K of any DDGA-recognizable assertion. Given some DDGA \mathcal{A} over the node alphabet Val^{Var} , we define a DDGA $\text{wp}(K, \mathcal{A})$ over the same alphabet such that $\text{wp}(K, \mathcal{A})$ first simulates the execution of K on the input graph and then operates like \mathcal{A} on the resulting graph. In other words, $\text{wp}(K, \mathcal{A})$ is an automaton that accepts a (Val^{Var}) -labeled graph G_λ if and only if \mathcal{A} accepts the labeled graph $G_{\lambda'}$ that one obtains after running the global command K on G_λ . Such an automaton can be effectively constructed because Var and Val are finite and the message exchange process in our algorithm framework is the same as for DDGAs. We can easily obtain $\text{wp}(K, \mathcal{A})$ by inserting one additional level of states before the first level of \mathcal{A} .

This brings us to the desired inference rule for $K = \text{each } v \text{ does } D \text{ end}$:

$$\frac{L_*(\mathcal{A}_\varphi) \subseteq L_*(\text{wp}(K, \mathcal{A}_\psi))}{\{\varphi\} K \{\psi\}}, \quad (\text{single round})$$

where \mathcal{A}_φ and \mathcal{A}_ψ designate DDGAs that recognize the properties specified by φ and ψ , respectively. This rule reduces the problem of deriving a Hoare triple for a single round to

the language inclusion problem of DDGAs. We know that the latter is decidable, because it can, in turn, be reduced to the emptiness problem of DDGAs on connected, undirected graphs, which we have shown to be decidable in Lemma 3.5.1. (This second reduction also relies on the fact that DDGA-recognizable graph languages are effectively closed under complementation and intersection, which holds by Lemma 3.3.5.)

For more complex global commands, we simply take over the inference rules from classical Floyd-Hoare logic, i.e., for any global commands K , K_1 , K_2 and DDGA-recognizable assertions φ , φ' , ψ , ψ' , θ , ξ , we have

$$\frac{\{\varphi\} K_1 \{\theta\} \quad \{\theta\} K_2 \{\psi\}}{\{\varphi\} K_1 K_2 \{\psi\}}, \quad (\text{sequence})$$

$$\frac{\{\theta \wedge \xi\} K \{\theta\}}{\{\theta\} \textbf{while } \xi \textbf{ do } K \textbf{ end } \{\theta \wedge \neg \xi\}}, \quad \text{and} \quad (\text{loop})$$

$$\frac{L_*(\mathcal{A}_\varphi) \subseteq L_*(\mathcal{A}_{\varphi'}) \quad \{\varphi'\} K \{\psi'\} \quad L_*(\mathcal{A}_{\psi'}) \subseteq L_*(\mathcal{A}_\psi)}{\{\varphi\} K \{\psi\}}. \quad (\text{strengthen/weaken})$$

We can now use this method to verify the FLOODMAX algorithm from Example 4.1.1.

4.2.1 Example (Verification of FLOODMAX).

An assertion-annotated version of the code is displayed in Algorithm 2. As is often the case in Floyd-Hoare proofs, verification is performed with the help of auxiliary variables, which may not be modified by the algorithm to be verified. We introduce an additional member variable $v.m_{\text{ini}}$ for each node v in order to be able to refer to the initial value of $v.m$, and, accordingly, our precondition is $\forall v (v.m = v.m_{\text{ini}})$. The algorithm satisfies partial correctness if, after termination, every node v knows the maximum initial value present in the graph, i.e., the postcondition is $\forall v (v.m = \max_u u.m_{\text{ini}})$.

The crucial part of the proof is finding a suitable loop invariant. Assertion θ in Algorithm 2 turns out to be adequate. It states that for each node v , the currently largest known number $v.m$ is bounded from below by the largest values known to its neighbors in the previous round and by its own initial value $v.m_{\text{ini}}$, and that there is some node in the graph that still retains its initial value. If we denote by K the loop body (lines 5 to 9) and by \mathcal{A}_θ a DDGA equivalent to θ , it is easy to see that $L_*(\mathcal{A}_\theta) \subseteq L_*(\text{wp}(K, \mathcal{A}_\theta))$, and consequently we can use the “single round” rule to derive the Hoare triple $\{\theta\} K \{\theta\}$. Note that our invariant does not even rely on the exit condition ξ of the loop. By the “strengthen/weaken” rule, we obtain $\{\theta \wedge \xi\} K \{\theta\}$, and then the “loop” rule allows us to derive $\{\theta\} \textbf{while } \xi \textbf{ do } K \textbf{ end } \{\theta \wedge \neg \xi\}$. It is also easy to see that the loop invariant θ follows from assertion φ , and that $\theta \wedge \neg \xi$ implies assertion ψ . Again, this can be expressed in terms of inclusions of DDGA-recognizable graph languages, and by the “strengthen/weaken” rule we get $\{\varphi\} \textbf{while } \xi \textbf{ do } K \textbf{ end } \{\psi\}$.

By proceeding analogously for the initialization part (lines 1 to 3), and then applying the “sequence” and “strengthen/weaken” rules, we can formally derive the desired Hoare triple

$$\{\forall v (v.m = v.m_{\text{ini}})\} \text{FLOODMAX} \{\forall v (v.m = \max_u u.m_{\text{ini}})\}.$$

Note that the postcondition follows from assertion ψ because we only consider graphs that

are undirected and connected. The first conjunct of ψ implies that all the nodes v have the same value for $v.m$, while the two remaining conjuncts ensure that this value is indeed the maximum initial value present in the graph.

Algorithm 2. FLOODMAX with Floyd-Hoare Annotations

```

{  $\forall v(v.m = v.m_{\text{ini}})$  }
1 each  $v$  does
2   |  $v.m_{\text{old}} \leftarrow 0$ ;
3 end
{  $\varphi$ :  $\forall v(v.m = v.m_{\text{ini}} \wedge v.m_{\text{old}} = 0)$  }
       $\xi$ 
4 while  $\exists v(v.m \neq v.m_{\text{old}})$  do
   | {  $\theta$ :  $\forall u, v(u \rightarrow v \Rightarrow u.m_{\text{old}} \leq v.m) \wedge \forall v(v.m \geq v.m_{\text{ini}}) \wedge \exists v(v.m = v.m_{\text{ini}})$  }
5   | each  $v$  does
6   |   | send  $v.m$  receive  $M$ ;
7   |   |  $v.m_{\text{old}} \leftarrow v.m$ ;
8   |   |  $v.m \leftarrow \max(M \cup \{v.m\})$ ;
9   | end
10 end
{  $\psi$ :  $\forall u, v(u \rightarrow v \Rightarrow u.m \leq v.m) \wedge \forall v(v.m \geq v.m_{\text{ini}}) \wedge \exists v(v.m = v.m_{\text{ini}})$  }
{  $\forall v(v.m = \max_u u.m_{\text{ini}})$  }

```

4.3 Prospects and Limitations

As mentioned at the beginning of this section, our adaptation of Floyd-Hoare logic is in principle not restricted to the toy language presented here. By replacing DDGAs with a more powerful class of (not necessarily finite-state) graph automata, we might directly obtain a variant of the framework in which we could formalize and verify more interesting distributed algorithms.

In order to be suitable for our purposes, an automaton model must

- be effectively closed under Boolean set operations,
- have a decidable emptiness problem, and
- be able to simulate a single synchronous round of any algorithm that can be specified in the corresponding formal language.

Hence, NDGAs and ADGAs cannot be used to extend this method (the former not being closed under complementation, the latter having an undecidable emptiness problem).

Besides covering a larger class of algorithms, a more expressive automaton model might also allow us to specify to-be-verified algorithms in a more natural way, less dependent on the global controller. With DDGAs, the controller has to compensate for the fact that we cannot, in general, provide DDGA-recognizable assertions on the nodes' knowledge about properties of the graph (such as the diameter in the FLOODMAX algorithm). If, on the other hand, we were able to express such assertions, the role of the controller could be reduced to simply checking whether all the nodes have terminated. Note that this would inevitably require an automaton model over infinite node alphabets, since the nodes would have to store information of unbounded size.

Our verification method also has a limitation that cannot be overcome by simply switching to another automaton model: it is only applicable to *synchronous* distributed algorithms. However, this might not be an issue in practice because any synchronous algorithm can be (automatically) converted into an asynchronous one using a synchronizer, as suggested by Awerbuch in [Awe85]. Thus, assuming the tool used for conversion is correct, a mechanical verification technique for synchronous algorithms also provides an indirect way of obtaining verified asynchronous algorithms. Since it is usually easier to design algorithms for synchronous systems, this seems like a practical approach.

4.4 Related Work

The method presented here is based on the simple observation that we can reason about a synchronous distributed algorithm as if it were a sequential algorithm whose elementary operations modify the global state of an entire graph. This approach has also been recently employed by Drăgoi et al. in [DHV⁺14], where they have considered fault-tolerant consensus algorithms operating in a synchronous setting that allows the topology of the communication graph to change nondeterministically in every round. (The FLOODMAX algorithm from Example 4.1.1 is a simple consensus algorithm.) In order to verify such algorithms, they have introduced a many-sorted, first-order-like logic with a very restricted syntax, in which they can reason about the global state of a graph and its underlying topology, as well as encode transitions between global states. For many consensus algorithms, that logic permits to formalize statements of the following form:

“If state G_λ satisfies invariant Inv and some condition on the topology of the graph, and additionally the algorithm permits a transition from G_λ to G'_λ , then the invariant Inv also holds in G'_λ .”

Here, G_λ and G'_λ have the same nodes, but possibly different edges. In this manner, Drăgoi et al. were able to formalize safety properties and termination of many consensus algorithms from the literature. The second part of their paper provides a semi-decision procedure for checking the validity of such verification conditions expressed in their logic. Furthermore, they could also identify a decidable fragment of that logic, which, for some algorithms, is already sufficient to prove correctness.

Acknowledgments

The author would like to thank Fabian Kuhn and Andreas Podelski (both from the University of Freiburg) for many comments and stimulating discussions. Their combined expertise was especially helpful for the part about verification of distributed algorithms.

References

- [Awe85] B. Awerbuch (1985), *Complexity of Network Synchronization*. Journal of the ACM, Vol. 32, No. 4, pages 804–823.
- [Büc60] J.R. Büchi (1960), *Weak Second-Order Arithmetic and Finite Automata*. Zeitschrift für Math. Logik und Grundlagen der Mathematik 6, pages 66–92.
- [Cho56] N. Chomsky (1956), *Three Models for the Description of Language*. IRE Transactions on Information Theory 2, pages 113–124.

- [CKS81] A.K. Chandra, D.C. Kozen, L.J. Stockmeyer (1981), *Alternation*. Journal of the ACM, 28, pages 114–133.
- [Cou90] B. Courcelle (1990), *The monadic second-order logic of graphs. I. Recognizable sets of finite graphs*. Information and computation 85, pages 12–75.
- [DHV⁺14] C. Drăgoi, T.A. Henzinger, H. Veith, J. Widder, D. Zufferey (2014), *A Logic-Based Framework for Verifying Consensus Algorithms*. Verification, Model Checking, and Abstract Interpretation (VMCAI 2014), pages 161–181.
- [Elg61] C.C. Elgot (1961), *Decision Problems of Finite Automata Design and Related Arithmetics*, Transactions of the American Mathematical Society 98, pages 21–51.
- [Fag75] R. Fagin (1975), *Monadic Generalized Spectra*. Zeitschrift für Mathematische Logik und Grundlagen der Mathematik, Vol. 21, pages 89–96.
- [Kle56] S.C. Kleene (1956), *Representations of Events in Nerve Nets and Finite Automata*. Automata Studies (C.E. Shannon and J. McCarthy, eds.), Princeton University Press, pages 3–42.
- [KVW12] I. Konnov, H. Veith, J. Widder (2012), *Who is afraid of Model Checking Distributed Algorithms?* Unpublished contribution to CAV Workshop (EC)².
<http://forsyte.at/download/ec2-konnov.pdf>
- [Lib04] L. Libkin (2004), *Elements of Finite Model Theory*. Springer.
- [LT00] C. Löding, W. Thomas (2000), *Alternating Automata and Logics over Infinite Words*. Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics, pages 521–535, Springer.
- [Lyn96] N.A. Lynch (1996), *Distributed Algorithms*. Morgan Kaufmann Publishers.
- [Ner58] A. Nerode (1958), *Linear Automaton Transformations*. Proceedings of the AMS 9, pages 541–544.
- [Rei14] F. Reiter (2014), *Distributed Graph Automata*. Master’s Thesis, University of Freiburg. [arXiv:1404.6503](https://arxiv.org/abs/1404.6503)
- [RS59] M.O. Rabin, D. Scott (1959), *Finite Automata and their Decision Problems*. IBM Journal of Research and Development 3, pages 114–125.
- [TATA08] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, M. Tommasi (2008), *Tree Automata Techniques and Applications*.
<http://tata.gforge.inria.fr>
- [Tho91] W. Thomas (1991), *On Logics, Tilings, and Automata*. In J.L. Albert, B. Monien, M. Rodríguez-Artalejo, eds., ICALP, volume 510 of Lecture Notes in Computer Science, pages 441–454, Springer.
- [Tra61] B.A. Trakhtenbrot (1961), *Finite Automata and the Logic of Monadic Predicates*. Doklady Akademii Nauk SSSR 140, pages 326–329 (in Russian).